

Implementation and Characterization of Three-Dimensional Particle-in-Cell Codes on MIMD Massively Parallel Supercomputers

P.M. Lyster, P.C. Liewer, V.K. Decyk* and R.D. Ferraro

Jet Propulsion Laboratory, California Institute of Technology, Pasadena, CA 91109

*Physics Department, University of California Los Angeles, CA 90024

Abstract

A three-dimensional electrostatic particle-in-cell (PIC) plasma simulation code has been developed on coarse-grained distributed-memory massively parallel computers with message passing communications. Our implementation is the generalization to three-dimensions of the General Concurrent Particle-In-Cell (GCPIC) algorithm. In the GCPIC algorithm, the particle computation is divided among the processors using a domain decomposition of the simulation domain. In a three-dimensional simulation, the domain can be partitioned into 1-, 2- or 3-dimensional subdomains ("slabs", "rods", or "cubes") and we investigate the efficiency of the parallel implementation of the push for all three choices. The present implementation runs on the Intel Touchstone Delta machine at Caltech; a Multiple-Instruction-Multiple-Data (MIMD) parallel computer with 512 nodes. We find that the parallel efficiency of the push is very high, with the ratio of communication to computation time in the range 0.3%-10.0%. The highest efficiency (>99%) occurs for a large, scaled problem with 64^3 particles per processing node (~134 million particles on 512 nodes) which has a push time of about 250 nsec per particle per time step. We have also developed expressions for the timing of the code which are a function of both code parameters (number of grid-points, particles, etc.) and machine-dependent parameters (effective FLOP rate, and the effective inter processor bandwidths for the communication of particles and grid-points). These expressions can be used to estimate the performance of scaled problems - including those with inhomogeneous plasmas - to other parallel machines once the machine-dependent parameters are known.

1. introduction

In particle-in-cell (PIC) codes¹, the trajectories of thousands to millions of particles are calculated as the particles move under the influence of fields computed self-consistently on a discrete grid. In plasma PIC codes, the electromagnetic forces on the charged particles are computed on the grid using source terms computed from the particles. There are typically tens or hundreds of particles per cell. Linking the particle orbit and the field-solve components of the computation are the gather/scatter steps that transform quantities back and forth between the particles and the grid. These computer codes with an excess of 10^6 simulation particles have been used to study ionized plasmas with many orders of magnitude more real particles. The ability to use these 'macro'-particles is the reason why problems have been made tractable on existing computers. However, there is always the need to model more accurately the position and velocity space structures that develop in plasmas, and also to reduce the particle noise. This can only be achieved by using more (up to 10^{10}) particles on the present and future generations of supercomputers. Distributed-memory massively-parallel supercomputers are a good candidate for this because of the inherent parallelism in following the trajectories of many particles.

The code we have developed is an extension of previous work on the General Concurrent Particle-In-Cell (GCPIC) algorithm^{2,3} to three dimensions. Particle-in-cell codes are made up of a particle-pushing component, which includes the two interpolations between the particles and the grid (the gather/scatter), and a field-solve. Most PIC codes are computationally bound by the particle-pushing step, with the field-solve typically taking less than 10% of the CPU time. In the GCPIC algorithm, the particle computation is divided among the processors by partitioning the domain and assigning a subdomain (the grid-points and the particles in it) to each processor. The GCPIC algorithm was designed to maximize the efficiency of the particle pushing step. The two primary considerations for any parallel algorithm are to balance the processor loads and to minimize interprocessor communication. To balance the loads, the GCPIC subdomains are created with equal number of *particles*, which, for non-uniform particle distributions, will have unequal numbers of grid-points. To minimize interprocessor communication, the particles and grid-points are in the same processor so that the gather/scatter operations can be performed with no interprocessor communication. In the course of the calculation, messages are needed to ensure computational consistency of the grid quantities across domain boundaries and to pass particles to new processors as they cross domain boundaries. The efficiency of this approach is enhanced for coarse-grained parallel computers where a large number of the particles and grid-points can be stored per processor leading to low ratios of communication time to computation time.

In the particle push portion of our GCPIC code, the communication burden is [a] the cost of passing particles between processor domains; and [b] the cost of communicating particle-moment data (density and current) between the edges, or 'guard-cells', of the processor domains; this is needed to ensure consistency of the resulting moment fields

(note, this is the only context in which the expression ‘guard-cell’ will be used in this paper). These are the only communication costs which will be analyzed in this paper. The field-solve portion involves other communication costs which are not considered in the work presented here. In the most general case, the GCPIC uses a second decomposition “ for the field-solve. For example, in the case where distributed fast Fourier transform (FFT) techniques are used, the secondary decomposition needs to have equal grid-points per processor, whereas particle load balancing considerations may dictate a non-uniform primary decomposition³. The communication burdens for the field-solve are [a] the cost of passing particle-moment and field quantities between the primary and secondary decomposition; [b] any costs of communicating field between the edges, or ‘guard-cells’, of the secondary domains; and [c] any costs of communications in a distributed (say FFT) field-solve. These costs can be considerable³, but are very highly dependent on the method used to solve the field equations and thus are not discussed here. A code with a finite difference field-solve and uniform particle distribution, for which only one decomposition is necessary, has the minimal communication costs.⁴

The code described here has been setup to be a testbed for the assessment of different particle and field decompositions, and algorithms. The knowledge we learn from this is expected to be used to find efficient decompositions for particle-in-cell codes that will study strongly inhomogeneous plasmas, such as a dense charged particle beam, as well as for the Numerical Tokamak projects. Our results may also be useful for particle hydrodynamic particle methods which have relatively few particles per cell but whose computational burden may still be high. Our testbed code is a three-dimensional electrostatic PIC plasma simulation code on Caltech's Intel Touchstone Delta (S 12 processors) machine and Gamma (64 processors) machines. These are coarse-grained memory Multiple-instruction-Multiple-Data (MIMD) supercomputers with message passing communication.

For the purposes of this paper it is necessary to distinguish between the dimensionality of the problem and the dimensionality of the domain decomposition. In this paper the lower case character ‘d’ refers to the former. Therefore a simple one-dimensional two-stream instability can be modeled with a $d=1$ code. We are mostly concerned with the performance of a three-dimensional code ($d=3$) since such codes will carry the forefront of current research. The dimensionality of the domain decomposition is specified by upper case ‘D’. In this paper we will describe primary decompositions into slabs ($D=1$), rods ($D=2$), and cubes ($D=3$). For example, in a one-dimensional slab decomposition a domain includes all y and z for a range of x ; in a two-dimensional “rod” decomposition a domain includes all z for a range of x and y ; for a three dimensional “cube” a domain includes a range of x , y , and z . In a spatially homogeneous maxwellian plasma the cubes have minimum surface to volume ratio and would minimize the number of particles that must be exchanged among processors. This situation may change for simulations with directed beams or currents.

For our electrostatic code on the Delta we can fit about 290,000 particles per node. Using 512 nodes on the Delta, we have run up to 150 million particles with a push time of 0.25

microseconds per particle per time step (electrostatic push with tri-quadratic (27 point) interpolations for gather/scatter).

We have studied the efficiency of the code for the three dimensionalities of the decomposition ($D=1, 2$ and 3) for a fixed problem size and varying number of processors. We found that the efficiency was generally insensitive to the dimensionality of the decomposition for the parameters studied and that even for these fixed problem-size runs, the efficiency was quite high ($>90\%$). We expect that for large-scale problems the dimensionality will matter. For example, for a homogeneous maxwellian plasma the most efficient decomposition in terms of the surface to volume ratio of the domains - and that would be the main factor that would determine communication overhead - is $D = 3$. Runs were also made in which only the number of particles or grid-points was varied. It was found that even for a case where 40% of the particles were exchanged at each time step, the ratio of communication to computation time was still low (5.5%). We also discuss the case where the nodes are fully utilized, that is, scaled problems that fill the physical memory of each processor, and whose size is proportional to the number of processors N_p . For these runs we find that the ratio of communication to CPU time is small (~ 3.0 - 3) and approximately independent of N_p . We have also analyzed runs with the higher compiler optimization -O4 and evaluated a push time of about 250 nsec per particle per time step.

We have also developed a general formulation for the timing-behavior of the code in terms of basic parameters -- the total number of processors N_p , the maximum memory per processor, the total number of grid-points N_g , the total number of simulation particles N , the effective rate of floating point operations (FLOPS) F , and the effective interprocessor bandwidths for the communication of particles B_{tr} , and grid-points B_{gv} . Our predictive formulation is found to be successful in predicting code performance on the Delta (with some caveats) and can be used to estimate the performance of problems on other parallel computers once the machine-dependent parameters have been determined.

Our approach represents an implementation that is applicable to coarse-grained memory (greater than 1 megabyte per processor) MLMD parallel computers. It is also specific to the set of problems where long range forces on the geometrical mesh dominate over the short interparticle forces - that is, the particle-mesh (PM) method. We note that particle codes that make use of Data Parallel coding techniques are under study^{6,7}. These codes often have the advantage of being more easily debuggable, but with present software are not always capable of using the flexible domain decomposition strategies that may be used in MIMD programming. Data Parallel algorithms have also been developed for the case where some correction to short range forces is needed, such as in some stellar dynamics codes⁸. Our paper is concerned with both the performance and the development of a simple algorithm that predicts the performance of the GPIC method for given machine parameters. Our analysis pertains directly to the case of uniform periodic plasmas; we will make some comments in Section 4 on the wide range applicability, particularly for non-uniform plasmas. We note other work in the field that relate to unstructured problems in a Data Parallel framework⁹, unstructured problems in MIMD¹⁰, and fast parallel tree codes for gravitational and fluid dynamical problems¹¹.

In Sec. 2, the parallel implementation of the code is presented. Section 3 presents performance results on the efficiency of the particle push for several sets of runs. Section 4 presents the derivation of the code performance expression and compares predictions with runs in Sec. 3. Our conclusions are presented in Sec. 5

2. Implementation of the Code

The code is implemented in parallel using the GCPIC algorithm^{2,3} which divides the particle computation among the processors using a domain decomposition: The computational domain is partitioned into subdomains with approximately equal numbers of particles and a subdomain, including grid-points and particles, is assigned to each processor. For uniform particle distributions, the subdomains are equal in size. Each processor also stores guard-cells around the periphery of its domain which are needed in interpolations between particles and grids. A GCPIC decomposition for a two-dimensional code is shown in Fig. 1.

There are four steps in the GCPIC particle push. The first step of the particle push is the particle update, e.g., the advance of the positions and velocities. For the electrostatic code the electric field \mathbf{E} provides the force on the particles and the field is interpolated from the grid to the position of the particle to obtain the force at the location of the particle. In the second step, after updating the positions and velocities, each particle's new position is checked and particles which have changed subdomains are exchanged with nearest neighbor processors. In the third step, the charge density, which is needed to advance the electric field, is computed by interpolating (depositing or scattering) the particle charges to the grid-points. After the trade, all particles are in the processor with the nearest grid-points, but, because the particle interpolation may involve several neighboring grid-points, a particle may deposit a portion of its charge to a grid-point which belongs to neighboring processor. Therefore, each processor contains enough guard-cells surrounding its subdomain to insure that the scatter is strictly local. In the fourth step, to obtain a consistent density for all interior grid-points, the values at the guard-cells are communicated to adjacent processors and superimposed at the appropriate cells near the boundary. The number of guard-cells (NG) needed will depend on the interpolation scheme used. Here, using a standard three-point quadratic scatter, a maximum of two guard-cells in each spatial dimension (NG=2) at each edge of the processor domain (or, equivalently, two layers of guard-cells on each face) are needed. For applications involving higher order scatter, or for gyrokinetic simulations¹² with particles that are spread over a gyroradius, more guard-cells may be needed. During the particle and grid trading steps, particles and grid-point data are buffered and exchanges in groups in order to minimize the effect of the start-up (latency) time of the communication calls.

In our code, the electric field is obtained using a spectral transform solution of Poisson's equation. In general, a distributed FFT would be used to form the fastest solution, and to minimize the memory usage. Since we are interested in the behavior with respect to the primary decomposition, we use a standard sequential FFT solve on each processor. The consistent density arrays are combined into a global array that is present in each processor,

and the global electric field is solved, Then, each processor uses only the field associated with its primary domain to push the particles. This completes the iteration cycle: particle updates, moment scatter, and field-solve.

The results presented in this paper correspond to a three. dimensional physical domain. Periodic boundary conditions are imposed in the x, y, and z directions. The three dimensional ($d=3$) physical domain can be partitioned into subdomain of 1-, 2-or 3- dimensions as shown in Figure 2. This shows a color rendering of select particles; the particles are colored by processor and there are. 64 processors, four in each dimension. Recall that we use the lower case letter d for the dimensionality of the physical domain and the upper case letter D for the dimensionality of the subdomains. Figure 2 shows a "slab" primary decomposition ($D=1$) where all the processors have a range of x for ally and z . Figure 2 also shows the square cross section "rod" decomposition ($D=2$) and finally Figure 2 shows the $D=3$ "cubes," which, for unequal numbers of grid-points in x , y and z , will not actually be cubes. Several different schemes are possible for exchanging particles and guard-cells in a $d=3$ code with $D=1, 2$, and 3 partitions. It is clear from Fig. 2, that the number of processors that must exchange particles and guard-cells will depend on the dimensionality D of the partitioning scheme used. For $D=1$ slab subdomains, particles and guard-cells will only need to be exchanged with neighbors to the left and right (For code accuracy, particles move less than a grid-point per time step). For $D=2$ and 3 , more processors will be involved. The method we have chosen is a simple extension of the method used in the $d=1$ code of Liewer and Decyk².

To exchange particles in the present code, each dimension is treated separately. Particles only need to be exchanged in dimensions which are partitioned, e.g., for a one-dimensional partition ($D=1$) of the x -axis, particle only need to be exchanged in the x direction. In the code, there is a loop over dimensions x , y and z . First, the global periodic boundary condition is imposed on the particles. Next, if this dimension is also a partitioned dimension, each particle's coordinate in the dimension is checked. Particles with a coordinate higher than the processor boundary in the this direction are placed in a "right"-going buffer and those with a coordinate lower than processor boundary are placed in a "left"-going buffer. After all particles have been checked, the left- and right-going particles are exchanged with the appropriate neighboring processors and the incoming particles are unpacked. Note that by buffering the particles, only two communication calls are needed per partition dimension to exchange all the particles; this minimizes the message-latency overhead. The pseudo code for the exchange follows.

```

c xleft(3) and xright(3): processor subdomain bounds>-ies in x, y and z
For i=1,3 Do
    Apply global boundary condition
    If i is a partitioned dimension then
        If (x < xleft(i)) pack in left-going buffer
        If (x ≥ xright(i)) pack in right-going buffer
        Send left-going buffer to left and receive from right

```

```

        Unpack buffer received
        Send right-going buffer to right and receive from left
        Unpack buffer received
    Endif
Enddo

```

The buffering technique used creates and then fills holes in the main particle array as particles are traded as described by Licwer and Decyk².

Thus, for a code partitioned in all three dimensions ($D=3$), particles are exchanged first in x , then y , and then z to move them to the appropriate processor. Particles that move diagonally are automatically handled properly in this scheme. By using this scheme, only two buffers are needed for exchanging the particles as opposed to the 26 that would be needed if all three dimensions were considered simultaneously. Similarly, only 6 communication calls are necessary as opposed to 26. For the usual case when particles move only to "nearest neighbor" domains, one pass through this loop moves particles to the proper processor. By allowing multiple passes through this loop, the code can handle the more general case that particles must move to a processor an arbitrary number of hops away. This is necessary if dynamic load balancing is employed and may be necessary for other decomposition strategies as well. To handle such case, the loop is repeated until all particles have been passed the necessary number of times to reach the appropriate processor.

The guard-cell exchange following the deposit is handled using the same basic strategy of communicating and filling in each dimension separately. Note that as long as **all** guard-cells are exchanged and added to appropriate interior and/or guard-cells, all of the corner cells, which are receiving deposits from several processors, are properly handled by this scheme. Moreover, since the communication is done inside a loop over dimensions, the buffers need only be sufficiently large to pass the maximum number of particles or grid-points that are traded in each dimension. Note also that by treating the communication separately in each dimension, the code is more flexible and can be more easily modified to run in one, two, or three dimensions depending on the needs of the physical problem being studied. Appendix A describes a potential problem for users of the present Intel compiler. This involves issues of machine precision, arithmetic, and the evaluation of conditional operators - the user of any new compiler should bear in mind that may become a problem. The problem is avoided by ensuring sufficient number of guard-cells (NG), and looping over the trade step to ensure that all particles are in their appropriate domains before the deposit step.

At present we are using Intel Touchstone Delta (512. node) and Intel Gamma (64 nodes) machines at Caltech. The smaller machine is useful during the development and debugging phase while the Delta is used for the production and performance runs. These machines are coarse-grained memory MIMD; the available memory is about 12.5 Megabytes per node for the Delta, and 15 Megabytes per node for Gamma, for our electrostatic code on the Delta

we can fit about 290,000 particles per node. Using 512 nodes on the Delta, we have run up to 150 million particles with a push time of 0.25 microseconds per particle per time step (electrostatic push with quadratic interpolations for gather/scatter). The Delta has at least 10 times the computing power of a single processor Cray Y-MP.

Our code has been written in Express-FORTRAN¹³. This is a single-program-multiple-data (SPMD) variation on multiple-instruction-multiple-data (MIMD) programming style. The code is written and compiled into a single object code. Each processor runs the object with a separate program counter and is loosely synchronized via message-passing and global communication functions. A standard interprocessor buffer 'exchange' call is used to trade particle and grid-point data. When the field-solve is invoked, a standard global buffer 'combine' operation is used to collect the separate density arrays into a single global array that is reproduced on each processor. The processor-geometry, that is, the way in which the processors map onto x space, is derived from Express automatic decomposition tools. Each processor is identified by a *physical* node number that is between 0, and N_p-1 , where N_p is the number of processors. The physical node numbers are used as arguments to the communication calls. The user specifies how many processors are in each dimension; for example a problem may have 8 processors in the x direction, 4 in the y , and 2 in z , giving a total of $N_p = 64$. Express calls provide the nearest-neighbor processors in the physical grid. The position of each processor relative to some origin node (usually the node that includes the origin of the x coordinate system in its primary domain) is specified by a *logical* node number. The logical node number starts as (0,0,0) at the origin and increments by integer amounts in each dimension. This is used to keep track of spatial offset for each processor relative to the x origin. The logical node numbers can be derived simply from the information obtained from the Express automatic decomposition tools.

The main particle loop involves the following subroutines, in order: push(), trade(), deposit(), solve(), and finally a diagnostic routine. The traditional particle gather step occurs in push(), and the scatter step is in deposit(). We employ three point per dimension (e.g., quadratic) interpolation for both gather/scatter. The code has been setup so that, when the dimensionality d of the code is changed, a switch in a global include file, changes the interpolation scheme automatically to handle $d=1$ (three point interpolation), $d=2$ (nine point interpolation), and $d=3$ (twenty seven point interpolation). Communication calls appear in trade(), where the particles are exchanged; in deposit, where guard-cells are exchanged to evaluate the correct particle density; and in solve() where the separate processor density arrays are combined into a global array.

To simplify the gather/scatter steps, array indices within a processor are based on the *global* indexing, e.g., field arrays are dimensioned as

```
dimension i_left_array_bound ( 3 ) , i_right_array_bound ( 3 )
dimension field_array ( i_left_array_bound ( 1 ) : i_right_array_bound ( 1 ) ,
1                i_left_array_bound ( 2 ) : i_right_array_bound ( 2 ) ,
2                i_left_array_bound ( 3 ) : i_right_array_bound ( 3 ) )
```

where $i_left_array_bound(i)$ [$i_right_array_bound(i)$] is the global index of this processor's lowest [highest] grid-point in the i th direction. These arrays boundaries are computed from the processor logical node number, the physical boundaries of the processor's subdomain and the number of guard-cells (NG). Thus, if a processor's subdomain starts at grid-point k in the z direction, $i_left_array_bound(3)=k-NG$. (Logical node number are used nowhere else in the code; physical node number are used in the communication routines.) By indexing in this way, array index references that apply during the particle gather/scatter steps are *exactly as they would be for a sequential code*; no processor offset are used in the gather/scatter steps. The array boundary values, $i_left_array_bound(i)$ and $i_right_array_bound(i)$, are passed as subroutine arguments.

We have also made the code flexible in the following areas: with small changes to an 'include' file we can change between a one ($d=1$), two ($d=2$), and three ($d=3$) dimensional code, as well as between slabs ($D=1$), rods ($D=2$), and cubes ($D=3$) for the primary (particle) decomposition of the grid. Also, depending on the needs of the gather/scatter interpolation scheme, we can easily change the number of guard-cells used.

The code uses dimensionless units. Thus the time step between successive cycles of the code is normalized in terms of the inverse plasma frequency $\omega_p^{-1} = (4\pi n_e e^2 / m_e)^{-1/2}$ where n_e is the average electron density, and m_e is the electron mass. Velocities are normalized by $\omega_p \Delta$ where Δ is the grid spacing. We have only one dynamical species present (electrons); the ions are assumed to be a charge neutralizing fixed background.

3. Performance Results

In this section, timing results from three sets of runs are presented and analyzed to study the parallel efficiency of the particle push portion of the code. The first set shows the dependence of the efficiency-on the dimensionality D of the domain decomposition and the scaling of the efficiency for a fixed size problem size, The second set shows the variation in efficiency for a scaled problem. The third set shows the dependence of the efficiency on the percentage of particles exchanged per processor per time step.

In our parallel code, performance is degraded by communication needed to trade particles after the position update and to exchange guard-cell information after the charge deposit. We define the communication time as

$$\tau_{comm} = \tau_{tr} + \tau_{gu}, \quad (1)$$

where τ_{tr} is the particle trade time and τ_{gu} is the guard-cell communication time. τ_{gu} incorporates the time to actually exchange the cells and the time to pack and unpack them, all of which are parallel costs. The trade time τ_{tr} incorporates both the time to actually exchange the particles and the time to unpack the traded particles into the particle array. Unpacking is actually not a communication cost, but it is a parallel cost. Typically the

unpacking time is between 20% and 30% of the actual exchange time, indicating the greater expense of interprocessor communication relative to the local memory access involved in the unpack. Generally τ_{tr} and τ_{gu} are about the same magnitude for these runs. These times are discussed more in Sec. 4.

We define the computation time τ_{CPU} as

$$\tau_{CPU} = \tau_{push} + \tau_{de} + \tau_{bc}, \quad (2)$$

where τ_{push} is the time to perform the particle force interpolation (gather) and velocity and position updates, τ_{de} is the time to deposit the particles' charge onto the grid (calculate the charge density), and τ_{bc} is the time to check the particles' position against the global and local boundary and pack the particles for trade as necessary. Note that the pack portion of τ_{bc} is, in fact, a parallel cost. However, it is much smaller than the boundary check which must be performed even in a sequential code. We use this definition in order to be able to compare measured times with predictions using the formulas derived in Sec. 4. The three costs in τ_{CPU} are all proportional to the number of particles per processor N/N_p where N is the number of particles and N_p is the number of processors.

Using these definitions, the total run time is $\tau = \tau_{comm} + \tau_{CPU}$ and we define the parallel efficiency of the code to be

$$\epsilon \equiv 1/(1 + \tau_{comm}/\tau_{CPU}). \quad (3)$$

For a one processor run with no communication, the efficiency is 100%. We have not included any degradation in efficiency due to processor load imbalance because the cases run have uniform particle distribution and equal processor computation loads.

a. Variation with Dimensionality of Domain Decomposition

First, the dependence of the parallel efficiency on the dimensionality of the domain decomposition D was investigated. A fixed three-dimensional ($d=3$) problem was run for $D=1, 2$ and 3 on various numbers of processors. This problem is chosen to be the maximum that fits on a single processor: $N=64^3=262,144$ particles, $N_g=32^3$ grid-points. Other parameters are $\Delta t=0.2$, and plasma thermal velocity $V_{ti}=0.812$ (we used a truncated Maxwellian distribution, which we had found useful for debugging purposes). For the single processor run, the size of the executable that is loaded onto each node is 10.2 megabytes. For more than one processor less memory is required since the fixed number of particles is divided amongst the processors. For a slab ($D=1$) partition we performed runs with 2, 4, 8, and 16 processors. For a partition into rods ($D=2$), the number of processors used was 4 ($=2^2$, i.e. 2 processors in x and 2 processors in y), 16 ($=4^2$), 64 ($=8^2$), and 256 ($=16^2$). For cubes ($D=3$), we used 8 ($=2^3$), 64 ($=4^3$), and 512 ($=8^3$).

The results for efficiency ϵ versus the number of processors N_p are plotted in Figure 3 for $D=1$ (open circles), $D=2$ (open triangles) and $D=3$ (open squares). The efficiency decreases with increasing number of processors as expected for a fixed size problem. The fact that it is so close to 1 indicates how small the communications burden is relative to CPU. Also plotted in Fig. 3 is the percent. of particles traded per processor per time step for $D=1$ (filled circles), $D=2$ (filled triangles) and $D=3$ (filled squares). Note that, for the $D=3$ case with $N_g=512$ and 11% of a processor's particles exchanged per time step, the efficiency is still 91.570. As expected, at a fixed number of processors, the percentage of particles traded is largest for $D=1$ because the slab subdomains have the largest surface to volume ratio. Figure 4, which plots $\tau_{\text{comm}}/\tau_{\text{CPU}}$ for the same cases for $D=1, 2$, and 3 (using circles, squares and triangles respectively), shows the steady increase in communication cost as the processor subdomains get smaller and relatively more particles and guard-cells are being traded. Note, however, that in both Fig. 3 and Fig. 4, the curves for all three values of D lie essentially on top of each other showing that the communication to computation ratio and, thus, the parallel efficiency is relatively insensitive to D for these parameters.

The communication to computation ratio for this code should scale as the surface to volume ratio, suggesting that $\tau_{\text{comm}}/\tau_{\text{CPU}}$ would decrease with D . The surface to volume ratio of a subdomain for arbitrary d and D is

$$S/V = 2 D N_p^{1/D} / N_g^{1/d}$$

where N_g is the total number of grid-points (so $N_g^{1/d}$ is the number in each direction) and $2D$ is the number of surface between processor subdomains. Since communication is only necessary across faces separating processor domains, the number of surfaces increases with D . Note that this expression for surface to volume ratio applies strictly to the case of periodic boundary conditions, and a uniform plasma where the domains are of equal size. At the end of Section 4 we discuss the situation of an inhomogeneous plasma. At this stage we note that for large problems with $N_p > 512$ the communications will be dominated by the interior domains - away from the boundaries. For small N_p this expression is somewhat artificial (in particular for $N_p = 1$ it is only a measure of S/V for a single periodic cube, however clearly no communications are needed in a single-processor sequential code). As we stated in the introduction, the goal of the paper is to show that the parallel performance of the 3D GCPI method is efficient for realistic problems ($N_p \gg 1$), and to show that it is predictable so long as some machine parameters such as flop rate and communication rate are precalculated. It can be seen that there are two reasons why $\tau_{\text{comm}}/\tau_{\text{CPU}}$ appears insensitive to D for small values of N_p . First, for small N_p , the exponential decrease ($N_p^{1/D}$) for increasing D is compensated for by the linear increase in the number of faces. Note that the largest number of processors used for $D=1$ was $N_p=8$ for which $S/V(D=1)=16/N_g^{1/d}$ vs. $S/V(D=3)=12/N_g^{1/d}$ and thus the ratio is 4/3. The second reason that the efficiency is insensitive to D for these runs is that as D increases, more communication calls are initiated since we exchange information separately in each dimension D of the decomposition. Thus, due to communication latency, the extra

communication calls needed for higher D counteracts the decrease in the total number of bytes communicated. For large problems on large numbers of processors we expect that the dependency of parallel efficiency on the dimensionality of the decomposition D to become more significant. This is already apparent for the percentage of particles traded for “D= 1 in Figure 3. Our problem was simply not large enough to demonstrate the behavior for $N_p \gg 16$. Similarly, the difference between D = 2 and D = 3 is not so apparent even for $N_p=512$, however for large problems on larger computers we expect the difference to be more significant.

Also shown in Fig. 3 are the theoretically calculated values for the percentage of particles traded for D=1, 2 and 3 (dashed lines); the agreement is excellent as expected for our known truncated Maxwellian distribution. The percentage traded is the number of particles traded per processor per time step N_{tr} divided by the number per processor N/N_p . The number of particles traded per time step will be the flux through each subdomain face multiplied by the surface area of the face times Δt times the number of faces. The number of subdomain faces is 2D since particle exchanges are only necessary through faces separating processor subdomains. Here, we assume $\Delta x = \Delta y = \Delta z = 1$ and that an equal number of processors ($N_p^{1/D}$) is assigned to each decomposition dimension as in Fig. 2. The total number of grid-points is N_g and $N_g^{1/d}$ is the number of grid-points in each direction where $d=3$ for the three dimensional cases in the paper. For a thermal species the flux of particles that cross a two-dimensional surface is given by $C_4 n v_{th}$, where v_{th} is the thermal (rms) velocity, n is the particle density ($n=N/N_g \Delta x^d$), and C_4 is a coefficient which is $1/(2\pi)^{1/2}$ for a homogeneous maxwellian distribution¹⁴ (for a directed beam we can replace v_{th} by the beam velocity, and $C_4 = 1$). Assuming square cross section rods and cubes, the surface area of each face of a processor domain is $N_g^{d-1/d} \Delta x^{d-1} / N_p^{(D-1)/D}$. The number traded is then, for arbitrary code dimensionality d and decomposition dimensionality D,

$$N_{tr} = 2 D C_4 (v_{th} \Delta t / \Delta x) N / (N_g^{1/d} N_p^{(D-1)/D}). \quad (4)$$

and the fraction of particles traded per processor per time step is

$$N_{tr} / (N/N_p) = 2 D C_4 (v_{th} \Delta t / \Delta x) N_p^{1/D} / N_g^{1/d}. \quad (5)$$

The predicted percentages shown in Fig. 3 were calculated using this equation. Note that the fraction traded is proportional to the surface to volume ratio, $S/V = 2DN_p^{1/D} / N_g^{1/d}$. There is, in fact, a systematic but small error not visible on the graph which we attribute to our use of a truncated maxwellian distribution. By comparing Eq (5) with the actual numbers of particles traded a measured value c_4 can be calculated. For all of the runs in the present series this quantity is roughly constant, and has an arithmetic mean of $c_{4m} = 0.43$ compared with $C_4 = 1/(2\pi)^{1/2} = 0.40$. In future calculations based on the above equations we will use the value c_{4m} in place of C_4 .

These first series of runs were performed with the default (-O 1) compiler option. For the single processor run, $\tau_{\text{push}} = 50.0$ seconds, τ_{bc} (essentially the time to check the particles for boundary conditions) = 5.75 seconds, and $\tau_{\text{dc}} = 16.6$ seconds. Note the relatively higher cost of computation plus gather/scatter compared with the check of boundary conditions. This is partially because the latter does not involve the loop cache inefficiency associated with grid-points from the random particle positions.

b. Efficiency for Scaled Problem

In the next series of runs, the efficiency was studied as the problem sized increased linearly with the number of processors so that the problem size in each node was fixed. All runs used the D=3 cube decomposition. The problem size on each node corresponds to the fixed problem size for the cases in Figs. 3 and 4 and is the largest problem that fits in one node: $N = 64^3 = 262,144$ particles and $N_g = 32^3$ grid-points. With this size problem in each of the 512 nodes, the total number of particles is $N = 512 \times \{ \approx 134 \text{ million}$ and the total number of grid-points is $N_g = 256^3 = 17 \text{ million}$. Intuitively this scaled problem should be highly efficient since on a percentage basis no more particles or grid-points are communicated as more processors are used. Note that this large number of grid-points could not fit on a single processor. Hence for these large runs, the field-solve (including the full N_g field array) was not applied. Only the local field to each domain was included in each processor, and that is how it should be for a distributed-memory massively-parallel algorithm. In this case we ran the code only for the purpose of obtaining numbers for particle pushing algorithm and associated communications.

The results are summarized in Fig. 5. The efficiency ϵ (squares) uses the left axis which goes from 0.95 to 1.0; the measured ratio $\tau_{\text{comm}} / \tau_{\text{CPU}}$ (circles) uses the right axis which goes from 0 to 0.05. Note that the efficiency is always >99% and is independent of the number of processors used as expected for the scaled problem runs. This illustrates the extremely high efficiency for this algorithm when the nodes are well utilized. In the runs with $N_p > 1$, there are about 3,400 particles traded per processor representing about 1.3% of particles. The predicted ratio $\tau_{\text{comm}} / \tau_{\text{CPU}}$ (dashed line) will be discussed in Sec. 4.

c. Variation with Particles per Processor

The cases in Figs. 3-5 showed very high efficiencies when a small portion of the particles were traded. To test the efficiency of the algorithm when a larger percentage of the particles were traded, a series of runs were made with a constant number of particles but a decreasing number of grid cells. For our three-dimensional code, Eq. (5) shows that the percentage of particles traded is proportional to $N_g^{-1/3}$, the number of grid-points in each dimension d , for other parameters held fixed. The results of these runs are shown in Fig. 6 plotted as a function of $N_g^{1/3}$, the number of grid-points in each direction. The top curve is the parallel efficiency (triangles) and the second curve is the fraction of particles traded per processor per time step (squares). Note that even when 40% of the particles in each

processor arc traded at each time step, the algorithm still has 95% parallel efficiency. Also plotted arc the measured $\tau_{\text{comm}}/\tau_{\text{CPU}}$ (circles) and the predicted $\tau_{\text{comm}}/\tau_{\text{CPU}}$ (dashed line). The predicted ratio will be derived and discussed in the next section.

4. Performance Analysis

a. Theoretical Discussion

Here, we derive expressions for the timing and efficiency in terms of code and machine dependent parameters. We have adopted a similar formulation to that of Lee and Azari¹⁵ and we loosely follow their nomenclature. However, in our work we have more closely parameterized the behavior of our code. The expressions derived here were used to generate the predicted curves in Fig. 5 and 6 and can be used to predict scaling behavior on other large massively parallel computers once the machine dependent parameters are known.

The code efficiency was defined in Eq. (3) in terms of the communication time τ_{comm} and the computation time τ_{cpu} , defined in Eq. (1) and (2,) respectively. First, we derive an expression for the communication time $\tau_{\text{comm}} = \tau_{\text{tr}} + \tau_{\text{gu}}$. Consider τ_{tr} , incorporating both the time to actually exchange the particles and the time to unpack the traded particles into the particle array. This trade time can be written as the number of bytes of information traded divided by an effective bandwidth (in bytes per second) for trading particle information B_{tr} . This effective bandwidth will be considerably lower than the rated channel hardware bandwidth because a) overhead due to communication startup (latency) adds some nonlinearity and b) it includes the user-time to unpack the particle information as well as the system-copy time. Thus, in general, B_{tr} will depend not only on the hardware, but, also on the size of the messages being passed because of the effects of latency. We write

$$\tau_{\text{tr}} = c_0 c_3 N_{\text{tr}} / B_{\text{tr}}, \quad (6)$$

where N_{tr} is the number of particles traded per processor per time step [Eq. (5)], $c_0 = 4$ bytes per word (single precision), and $c_3 = 7$ words of information per particle (3 space, 3 velocity and a tag). Similarly, τ_{gu} , which incorporates the time to actually exchange the cells and the time to pack and unpack them, can be written as the number of bytes of guard-cell information exchanged (bytes per word times 1 word per guard-cell) divided by the effective bandwidth for exchanging guard-cell information B_{gu} . Thus, with N_{gu} the number of guard-cells, we write

$$\tau_{\text{gu}} = c_0 N_{\text{gu}} / B_{\text{gu}}. \quad (7)$$

As for B_{tr} , B_{gu} will be less than the rated hardware bandwidth and will depend on message size. The number of guard-cells that must be communicated is the number of guard-cells per face (it, the number of guard-cells in each dimension multiplied by the number of grid-points per face) multiplied by the number of faces ($= 2D$). The number of grid-points per face is $N_g^{(d-1)/d} / N_p^{(D-1)/D}$, giving

$$N_{gu} = 2 D NG N_g^{(d-1)/d} / N_p^{(D-1)/D} \quad (8)$$

where NG is the number of guard-cells in each dimension at each domain boundary. As discussed in Sec. 2, NG is generally determined by the interpolation scheme used and is chosen so that the interpolations can be done locally. For the quadratic interpolation scheme used in the present code, NG=2.

The effective bandwidths B_{tr} and B_{gu} in Eqs. (4-5) are machine dependent parameters that must be determined empirically. Ideally, these bandwidths would be constant for a given computer, but, because of the communication overhead associated with sending a message (latency), we expect an additional dependence on message size. Moreover, because τ_{tr} and τ_{gu} also include some buffer packing and unpacking, they will, by definition, be lower than the rated channel communication bandwidth. We will determine these for the Delta using the runs in Sec. 3a and b (Figs. 3-4).

Note that for a code which uses a local finite difference field-solve and uses the primary decomposition to partition the field-solve, the only additional communication cost will be the time to exchange the field guard-cell information. Thus this analysis can be extended to include communication costs of such a field-solve by simply multiply Eq. (7) by the total number of field quantities.

Next, we derive an expression for the computation time $\tau_{CPU} = \tau_{push} + \tau_{de} + \tau_{bc}$, where τ_{push} is the time to perform the particle force interpolation (gather) and orbit update, τ_{de} is the time to deposit (scatter) the particle charge to the grid, and τ_{bc} is the time to check the particles global and local boundary and pack the particles for trade as necessary. With the exception of packing the particles to be traded (a negligible contribution), these times are proportional to the number of particles per processor N/N_p and we parameterized this time as

$$\tau_{CPU} = C_7 (N/N_p) / F, \quad (9)$$

where C_7 is a constant taken to be the total number of FLOPS involved in the particle and charge density update per time step and F is an effective flop-rate for the computation which has units FLOPS and, as above, is an 'effective' rate whose value will be less than the rated FLOPS per processor. For our code in three dimensions ($d=3$), we estimate $C_7 \approx 404$ (294 in the gather and orbit update, and 110 in the scatter determined by counting + and * each as 1 FLOP). Included in τ_{CPU} is also the memory access time associated with the particle pack and boundary condition checker in subroutine trade(). Thus, although there is a small cost incurred because of parallel processing, we characterize it together with other processes whose cost is proportional to the number of particles per processor N/N_p . We note that in our implementation the particle boundary conditions are applied at the same time as the packing process and the comparison with the primary domain boundaries uses almost the same FORTRAN code as the comparison with the global physical domain

boundaries. Equation (9) defines the effective FLOP rate F which must be determined for each parallel computer; we will use the runs in Sec. 3a to evaluate it for the Delta.

The quantity $\tau_{\text{comm}} / \tau_{\text{CPU}}$ can then be written as

$$\tau_{\text{comm}} / \tau_{\text{CPU}} = \tau_{\text{tr}} / \tau_{\text{CPU}} + \tau_{\text{gu}} / \tau_{\text{CPU}} \quad (10)$$

where

$$\tau_{\text{tr}} / \tau_{\text{CPU}} = (2 D c_0 c_3 c_4 c_5 / c_7) N_g^{-1/d} N_p^{1/D} F / B_{\text{tr}}, \text{ and} \quad (11)$$

$$\tau_{\text{gu}} / \tau_{\text{CPU}} = (2 D c_0 N_g / c_7) N_g^{(d-1)/d} N_p^{1/D} N^{-1} F / B_{\text{gu}}, \quad (12)$$

where $C_5 = v_{\text{th}} A_t / A_x$. Note that both ratios depend on the surface to volume ratio of the subdomains, $S/V = 2DN_p^{1/D} / N_g^{1/d}$. While equations (1)-(12) parameterize the timing of the code, at this point the coefficient F , as well as B_{gu} and B_{tr} are unknown. The fixed problem runs from Sec. 3.a are used to determine them empirically.

Figure 7 shows the effective FLOPS rate F , normalized to one megaFLOPS, determined from Eq. 9 using the *measured* τ_{CPU} for the runs of Sec. 3.a for $D=1, 2$ and 3 plotted as circles, triangles and squares respectively. The plot shows that F is approximately constant, independent of the number of processors N_p and the dimensionality of the domain decomposition D . Interestingly F shows a small increase with N_p . We also noticed that for the first 30 time steps of these runs τ_{CPU} showed a steady increase of about 8%. This can be traced to a gradual diminishment of cache efficiency as particles, which were initialized uniformly in space, are randomized due to their thermal motions. The times that we have used in this study are averaged over the last five time steps of a thirty time step run. This increase is an interesting insight into well known local memory access inefficiency of the gather/scatter process -- the τ_{CPU} is initially smaller because we have effectively sorted the particles relative to the grid at time step 1. The increase in τ_{CPU} occurs at a different rate for increasing N_p , that is, as the domain of each processor becomes smaller. The differing rate of increase of τ_{CPU} is responsible for the apparent small increase in F .

Figure 8a shows the effective trade bandwidth B_{tr} , and Fig. 8b the number of particles and bytes traded for $D=1, 2$, and 3 plotted as circles, triangles and squares respectively. The dominant behavior is the drop in communication speed as the number of bytes traded decreases for $D=2$ and 3 . This is the anticipated degradation due to the communication latency (start-up) cost as the size of buffers decreases. Intel¹⁶ state that the latency for standard node-to-node communications becomes significant when the number of bytes falls below 10^3 . We observe similar behavior here for the Express buffer-exchange routine that we use. Note that B_{tr} contains a particle buffer-unpack cost that is between 20% to 30% of the overall particle-trade cost, but its contribution will vary approximately linearly with the number of bytes traded. Note also that the bandwidth is lower for $D=3$ than $D=2$ when the same number of bytes are exchanged. This is because the number of messages actually

passed is proportional to D because we exchange in each decomposition direction separately, as discussed in Sec. 2, and thus effects of latency are increased.

Figure 9a shows the effective bandwidth for guard-cell communication B_{gu} , while Fig. 9b shows the number of guard-cells (and bytes) traded for $D=1, 2$ and 3 plotted as circles, triangles and squares respectively. For $D=2$ and 3 as above, B_{gu} falls off with the number of bytes communicated. (The observed slight increase for $D=1$ occurs because the code was written to implement periodic boundary conditions in the same loop that communications are performed. The actual number of cells traded per processor remains constant for $D=1$.) Note that B_{gu} is smaller than B_{tr} , due in part to the fact that B_{gu} contains both a guard-cell buffer pack and unpack costs while B_{tr} contains just the particle unpack costs and in part due to the on-average smaller number of bytes traded. B_{gu} does not show as large a difference between $D=2$ and $D=3$ presumably because of the stronger dependence on the packing and unpacking which is independent of D .

b. Comparison with Measured Code Times

Using the results of the above analysis and the determination of the effective bandwidths and their dependence on message size, we now calculate the predicted ratios τ_{comm}/τ_{CPU} and τ_{gu}/τ_{tr} for other runs and compare it with measured ratios.

The predicted τ_{comm}/τ_{CPU} was calculated for the scaled problem runs (Fig. 5) using Eqs. (10-12) with an effective flop rate of $F=1.47$ (Fig. 7) and using $B_{tr}=0.96$ and $B_{gu}=0.49$. These bandwidths are found from Figs. 8 and 9 extrapolating to the value appropriate to the number of bytes being exchanged, calculated using Eqs. 4 and 8. Both measured and predicted ratios of τ_{comm}/τ_{CPU} are shown in Fig. 5 and the agreement is excellent. For these scaled problem runs, the predicted ratio is constant as measured. We note that the particle trade and the guard-cell exchange each compose half of the estimated ratio. The scaled runs in Fig. 5 were performed with the -01 compiler option. When the -04 option is used the effective FLOP rate F increases by a factor of around 2.8. With this option the scaled runs perform with a push time of around 250 nsec per particle per time step.

The theoretical analysis was also used to predict the performance for the runs in Fig. 6. For this case, $N_p=512$, $N=512^3$, and only N_g was varied. Using Figs. 7 and 8, the quantities $F=1.47$, and $B_{tr}=0.96$ were used for all N_g since these depend only on N_p . However, the amount of guard-cell information exchanged varied with N_g and, thus the value of B_{gu} , interpolated from Fig. 9, varied with N_g . (The values used were $B_{gu}=0.31, 0.66, 0.96, 0.96$ for $N^{1/3}=64, 128, 256, 512$ respectively.) The comparison between the estimated and measured communication ratios, τ_{comm}/τ_{CPU} , shown in Fig. 6 is quite good. We note that the run with $N_g^{1/3}=32$ corresponds to the $N_p=512$, $D=3$ fixed problem case in Sec. 3a (Fig. 3&4) in all parameters except the total number of particles. In the former 27,000 particles were traded on average (1170), while in the fixed problem run 55 particles were traded (also 11%). However in the former the communication ratio is 0.0138 while for the fixed problem run it is 0.085. The improvement is due to the higher effective bandwidth B_{tr} as the larger number of particles traded reduces the effect of communication latency.

As a final example, in Fig. 1() we have also compared the measured and estimated (dashed line) values for the ratio τ_{gu} / τ_{tr} for a set of runs in which only the number of particles was varied. For these runs, $N_p=512$, $D=3$ and $N_g^{1/3} = 32$. N was varied from $N^{1/3} = 64$ to 512. We note that the run with $N^{1/3} = 64$ corresponds to the $N \geq 512$, $D=3$ fixed problem case in Sec. 3a (Fig. 3&4). As expected, the ratio is very sensitive to the number of particles and the agreement between the measured and estimated values is excellent.

c. The Case of Inhomogeneous Plasmas

The situation where the density of particles is non-uniform adds to the complexity of the resulting code, however it does not necessarily alter the performance from what we have described to this point. Since the CPU cost is so dominant, load balancing requires that the ratio of particles to processors be a constant everywhere in the domain. The software must be altered to allow for domains of varying size such that the ratio N_i/N_{pi} is a constant. We have added a new subscript here 'i' to distinguish between regions where the density is changing -- therefore, $N_i(N_{pi})$ is the number of particles (processors) assigned to region 'i' where the density is N_i/N_{gi} particles per local grid point. What changes is the ratio of the time to communicate particles to the time to communicate grid points. From Eqs. (11) and (12) we can easily show that

$$\tau_{tri} / \tau_{gui} = (c_3 c_4 c_5 / NG) (B_{tr} / B_{tr}) N_i / N_{gi} \quad (13)$$

That is, for $N_i/N_{pi} = \text{constant}$, the ratio of cost to communicate particles versus the cost to communicate grid points is given by some constant times the local density of particles per grid point. For the parameters used in the first set of (uniform density) runs in Section 3a this ratio is $\tau_{tr} / \tau_{gu} \sim 0.1(N/N_g) = 0.8$; clearly for eight particles per grid point this cost ratio is well balanced. For the case of strongly inhomogeneous plasmas we would expect this ratio to become unbalanced in some regions. However, while the CPU cost is so dominant, this imbalance is not important. Most of the results of this paper have been quoted for the unoptimized compiler option (-O 1). Hence any weakness in the above argument would not have been exposed. However in the case of highest optimization (-O4), our conclusion does not change. Only when the processor speed is increased considerably with respect to the communication speeds for MIMD machines - and we do not expect this to happen in the near future - would our conclusions be altered. We emphasise our prior point that we have provided a consistent framework whereby the performance can be assessed in terms of the simply derived machine parameters F , B_{gu} , B_{tr} , and memory per processor.

4. conclusions

We have developed a three-dimensional electrostatic particle-in-cell (PIC) plasma code for MIMD massively parallel supercomputers such as the Intel Delta and Gamma. The particle push portion of the code, which accounts for most (-90%) of the computation, is implemented using the GCPIC algorithm² in which the particle computation is divided

among the processors using a domain decomposition.¹ This work is an extension to three dimensions of previous work using the GCPIC algorithm^{2,3}. In the parallel code, inter-processor communication is necessary at two stages in the particle push. After the particle Positions are updated, particles which have left a subdomain must be passed to the appropriate processor. After the charge deposition, guard-cell information must be exchanged. Very high parallel efficiencies are found for the particle push, with the efficiency >99% for cases when the nodes are fully utilized. The efficiency was found to be > 95% even when up to 40% of the particles per processor per time step were traded. Because of this high efficiency even for large percentages of the particles traded, the GCPIC algorithm should also be efficient for non-uniform particle distributions where subdomains in regions of high particle density will be relatively small with large fractions of the particle leaving each time step.

In a three-dimensional code, the simulation domain can be partitioned in 1, 2 or 3 dimensional subdomains (slabs, rods, cubes). The ratio of communication to computation scales as the surface to volume ratio of the subdomains and thus a three-dimensional partition should be optimum. The efficiencies of the particle push was studied as a function of dimensionality of the subdomains. For the parameters studied, the efficiency was relatively insensitive to the subdomain dimensionality. This resulted from an increasing number of communication calls with higher subdomain dimensionality counterbalancing the surface to volume ratio effect. Only for small numbers of processors (≤ 8) will these effects cancel. For large number of processors, the surface to volume effect will dominate and the three-dimensional partitions will be more efficient.

Equations for predicting code performance based on code and machine dependent parameters were also developed. Machine dependent parameters were found from one set of runs. Predictions using the equation were in excellent agreement with measured performance in other sets of runs. Thus these formulae can be used to predict performance on other parallel computers once the machine dependent parameters have been established from a set of "calibration" runs. We expect that these formulae will be useful for the case of inhomogeneous plasmas with non-periodic boundary conditions, as discussed in Section 4.

Acknowledgments

We would like to acknowledge useful discussions with Mark Kiefer, David Forslund, Edith Huang, Bob Kares, Erik Matson (Jet Propulsion Laboratory Scientific Visualization Laboratory), David Payne (Intel), and Patti Sparks. This work was carried out at the Jet Propulsion Laboratory, California Institute of Technology, under a contract with the National Aeronautics and Space Administration. The research was supported in part by Sandia National Laboratory, Albuquerque and in part by NSF under Cooperative Agreement CCR-88809615. The computations were performed on the Intel Touchstone Delta parallel supercomputer operated for the Concurrent Supercomputing Consortium by the California Institute of Technology; access was provided by NASA.

Appendix A. Discussion of Machine Precision and Compiler Issues.

A common operation in any particle code involves the addition of an integer or half integer to a particle position. This happens when periodic boundary conditions are applied, and during the calculation of array indices during gather/scatter steps. Consider firstly the application of periodicity. At present the global physical domain is taken to be $-0.5 \leq x_i < N_i - 0.5$, where subscript i refers to spatial dimension, x_i is the particle position, and N_i is the number of grid-points in that dimension. Consider a particle whose position is $-0.5 - \epsilon$ where ϵ is close to machine precision, but sufficiently finite that FORTRAN `if(x < -0.5)` is true. In that case the periodicity condition is $x_i = x_i + N_i$, which should be evaluated to $N_i - 0.5 - \epsilon$. However the present i860 compiler on the Delta evaluates this to exactly $N_i - 0.5$. This number does not satisfy the condition of being inside the physical domain. The solution is to make sure that the boundary conditions are applied twice, so the particle is returned to **-0.5**. The problem has another manifestation in that a particle at position $N_i - 0.5 - \epsilon$ will, for the same reasons, incorrectly identify its nearest grid-point as `int(x + 1.5) = N_i + 1`, thus incurring the possibility of a segmentation violation during the scatter step. This problem is handled by ensuring the at least two guard-cells are used at both ends of each domain for three point interpolation. In general, the physical domain boundaries do not need to lie on half integer (or even integer) positions, however some care is always needed to ensure enough guard-cells are used.

References

- (1) C.K. Birdsall and A.B. Langdon, *Plasma Physics via Computer Simulation*, (McGraw-Hill, New York, 1985); R. W. Hockney and J. W. Eastwood, *Computer Simulatioon using Particles*, (McGraw-Hill 1981).
- (2) P.C. Liewer and V.K. Decyk, "A General Concurrent Algorithm for Plasma Particle-in-Cell Simulation Codes", *J. Comput. Phys.* 85, 302 (1989).
- (3) R.D. Ferraro, P.C. Liewer, and V.K. Decyk, "Dynamic load Balancing for a 2D Concurrent Plasma PIC Code", *J. Comput. Phys.* 109, 329 (1993).
- (4) J. Wang, P.C. Liewer, V.K. Decyk, "A Parallel Three-Dimensional Electromagnetic PIC Code for MIMI Parallel Computers," (submitted for publication, 1994).
- (5) J.M. Dawson, V.K. Decyk, R. Sydora, and P.C. Liewer, "High-Performance Computing and Plasma Physics," *Physics Today*, March, p. 64 (March 1993).
- (6) D.W. Walker. Particle-in-Cell Plasma Simulation Codes on the Connection Machine, *Computing Systems in Engineering*, 2, 307 (1991).
- (7) R. G. Hohlfeld, N. F. Comins, D. Shalit, P. A. Shorey, and R. C. Giles, Implementaion of Particle-in-Cell Stellar Dynamics Codes on the Connection Machhw-2, *J. Supercomputing* 7, 417 (1993).
- (8) T. Theuns, Parallel P3M with Exact Calculation of Short Range Forces, *Oomp. Phys Comm.* 78, 328 (1994).
- (9) Z. Johan, T. J. R. Hughes, K. K. Mathur, and S. L. Johnsson, A Data Parallel Finite Element Method for Computational Fluid Dynamics on the Connection Machine System, *Computer methods in Applied Mechanics and Engineering* 99, 113 (1992).
- (10) H. D. Simon, Partitioning of Unstructured Problems for Parallel Processing, *Computer Systems in Engineering* 2, 135 (1991).
- (11) J. K. Salmon, M. S. Warren, G. S. Winckelmans, Fast Parallel Tree Codes for Gravitational and Fluid Dynamical N-Body Problems, *The International Journal of Supercomputing Applications* 8, 129 (1994).
- (12) W.W. Lee, *J. Comput. Phys.* 72, 243 (1987).
- (13) J. Flower, and A. Kolawa, *Physics Reports* 207, Nos. 305, p. 291 (1991) North Holland.

- (14) F. Reif, *Fundamentals of Statistical and Thermal Physics*, (McGraw-Hill, New York, 1965)
- (15) N.G. Azari and S.-Y. Lee, "Hybrid Task Partitioning for Particle-in-Cell Simulation on Shared Memory Systems", Proceedings of International Conference on Distributed Computing Systems, pp526-533, Dallas, TX, May 1991. S.-Y. Lee and N.G. Azari, "Hybrid Task Decomposition for Particle-in-Cell Methods on Message Passing Systems", Proceedings of international Conference on Parallel Processing, vol. 111, pp141-144, St. Charles, IL, August 1992.
- (16) David Payne, Private Communication, (1992).

Figure Captions

- Figure 1 GCPIC domain decomposition for a two-dimensional code. Each processor has a sub-domain of the grid, the particles in it, and guard-cells around the perimeter.
- Figure 2 Possible GCPIC decompositions of a three-dimensional grid. One dimensional “Slabs” ($D=1$), two-dimensional “rods” ($D=2$), and three dimensional “cubes” ($D=3$). Showing particles colored by processor.
- Figure 3 Plot of measured parallel code efficiency (top curve and left axis) and percentage of particles traded (lower curves and right axis) versus $\log_2 N_p$ for $D=1$ (circles), 2 (triangles), and 3 (squares) for fixed problem size runs. The predicted percentages of particles traded are plotted as dashed lines.
- Figure 4 Communication to computation ratio $\tau_{\text{comm}}/\tau_{\text{CPU}}$ versus $\log_2 N_p$ for $D=1$ (circles), 2 (triangles), and 3 (squares) for fixed problem size runs.
- Figure 5 Parallel code efficiency (left axis) and $\tau_{\text{comm}}/\tau_{\text{CPU}}$ (right axis) versus number of processors N_p for scaled problem inns. The problem size per node is fixed so the problem size increases with N_p . The predicted ratio $\tau_{\text{comm}}/\tau_{\text{CPU}}$, derived in Sec. 4, is plotted as a dashed line.
- Figure 6 Parallel code efficiency (triangles), fraction of particles traded N_t/N (squares), measured $\tau_{\text{comm}}/\tau_{\text{CPU}}$ (circles) and predicted $\tau_{\text{comm}}/\tau_{\text{CPU}}$ (dashed line) versus the number of grid-points in one direction $N_g^{1/3}$. The expression for the predicted $\tau_{\text{comm}}/\tau_{\text{CPU}}$ is derived in Sec. 4.
- Figure 7 Effective FLOPS rate F versus $\log_2 N_p$ for $D=1$ (circles), 2 (triangles), and 3 (squares) derived from the fixed problem runs in Figs. 3 and 4.
- Figure 8 (a) Effective bandwidth for trading particles B_t versus $\log_2 N_p$ for $D=1$ (circles), 2 (triangles), and 3 (squares) for fixed problem size runs. (b) Number of particles and bytes traded versus $\log_2 N_p$ for $D=1$ (circles), 2 (triangles), and 3 (squares) for fixed problem size inns. The bandwidth drops as the number of bytes traded drops due to communication latency (overhead).
- Figure 9 (a) Plot of effective bandwidth for trading Guard-cells B_{gu} versus $\log_2 N_p$ for $D=1$ (circles), 2 (triangles), and 3 (squares) for fixed problem size runs. (b) Plot of the number of guard-cells traded versus $\log_2 N_p$ for $D=1$ (circles), 2 (triangles), and 3 (squares) for fixed problem size.
- Figure 10 Measured (x) and estimated (dashed line) Values for the ratio τ_{gu} / τ_t vs. number of particles N for runs with $N_p=512$, $D=3$ and $N_g^{1/3}=32$.

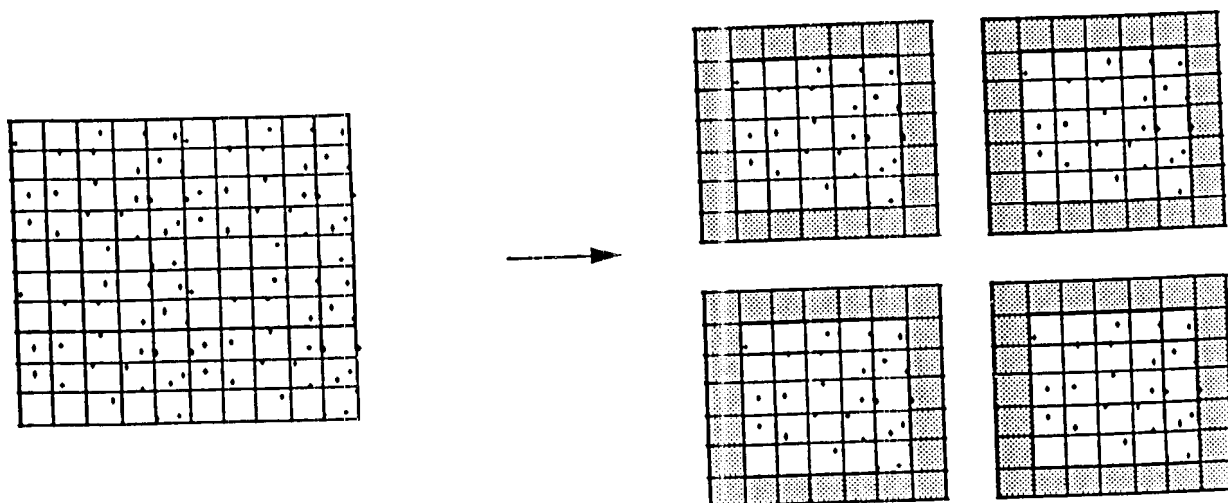


Figure 1

One Dimensional Decomposition

time: 0.2

Two Dimensional Decomposition

time: 0.2

Three Dimensional Decomposition

time: 0.2

Figure 2

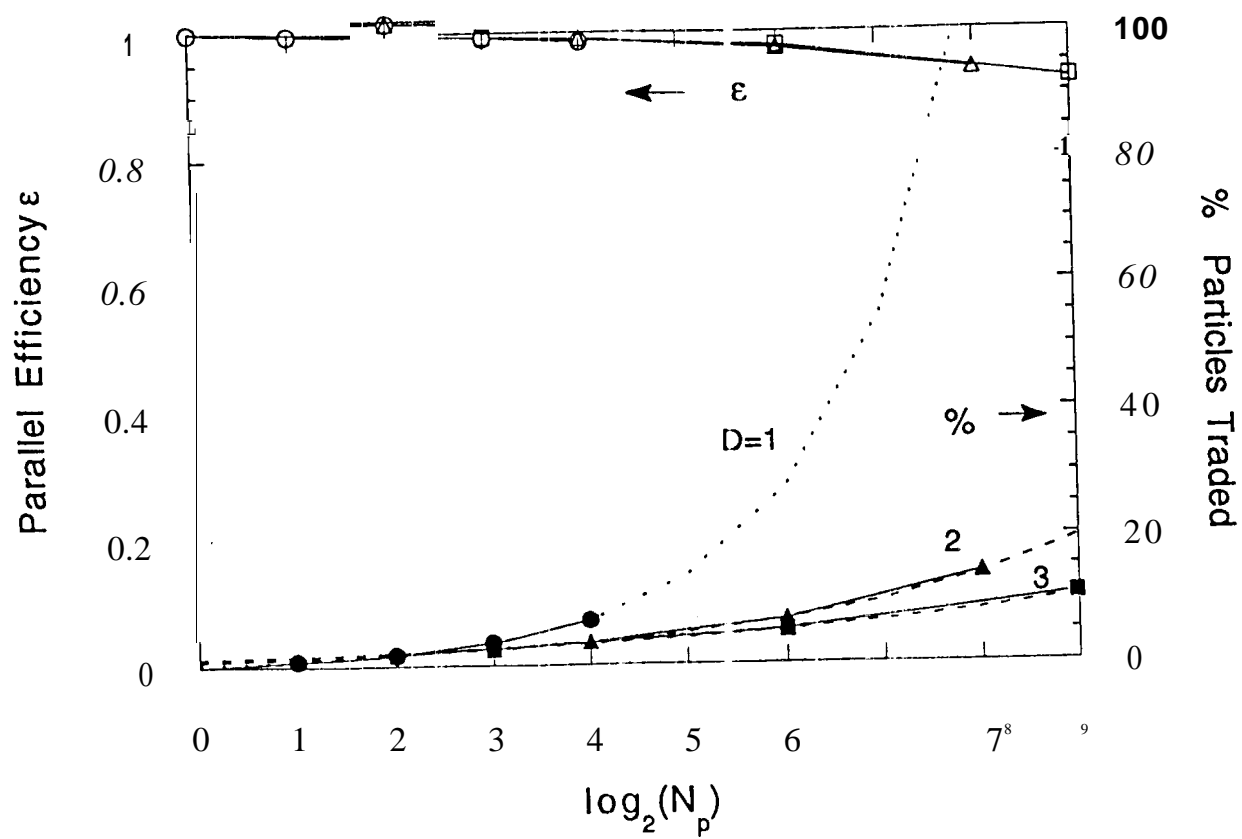


Figure 3

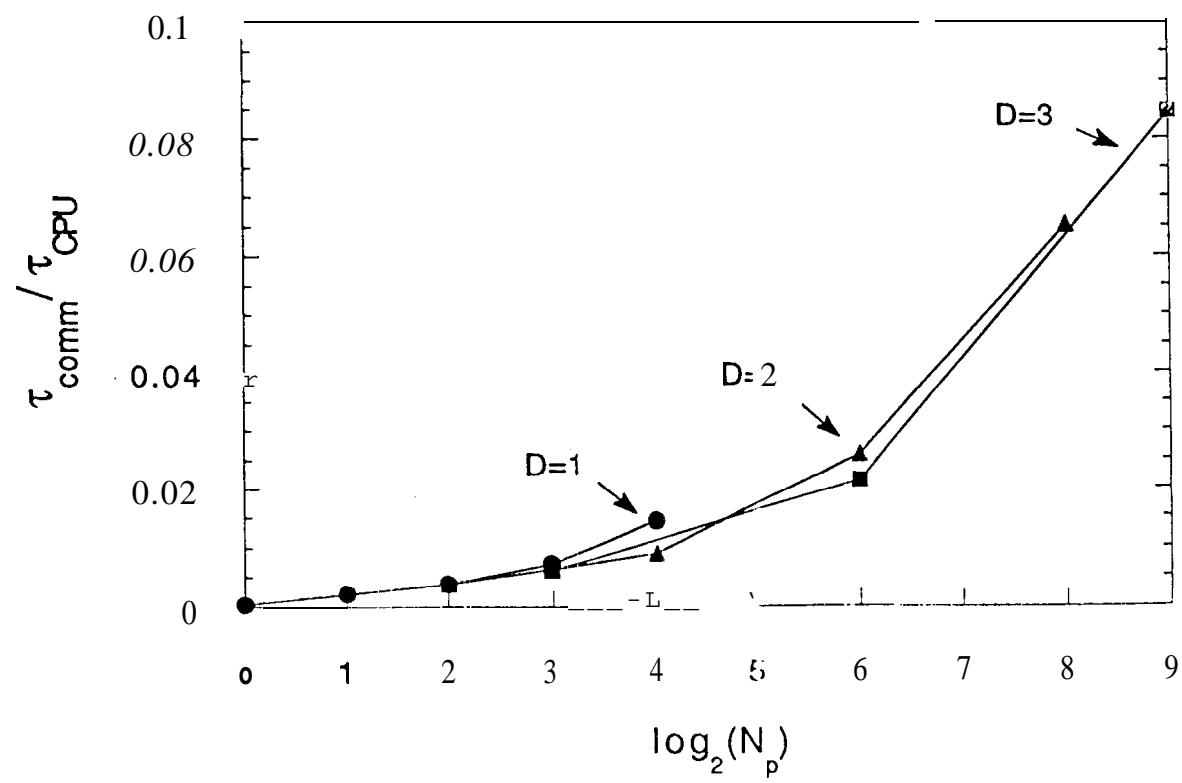


Figure 4

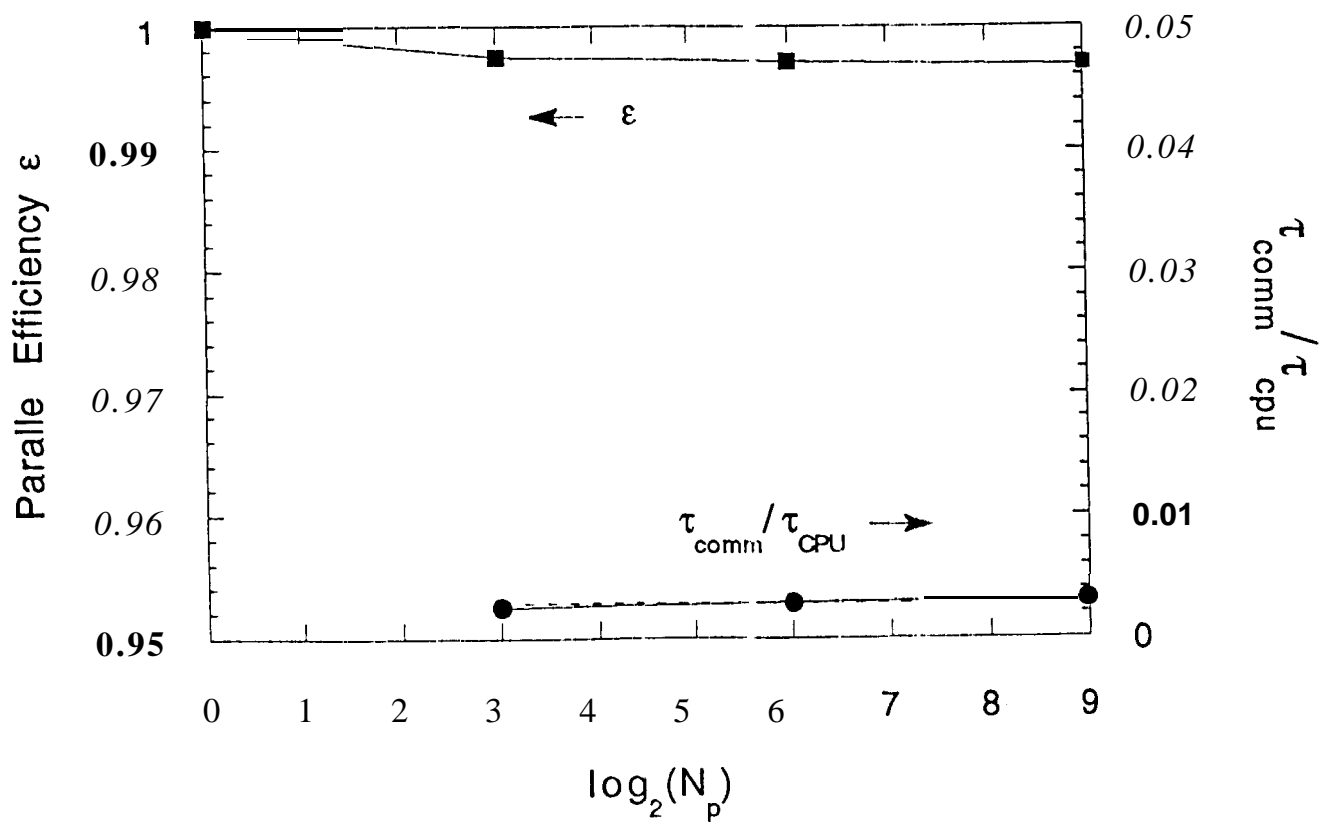


Figure 5

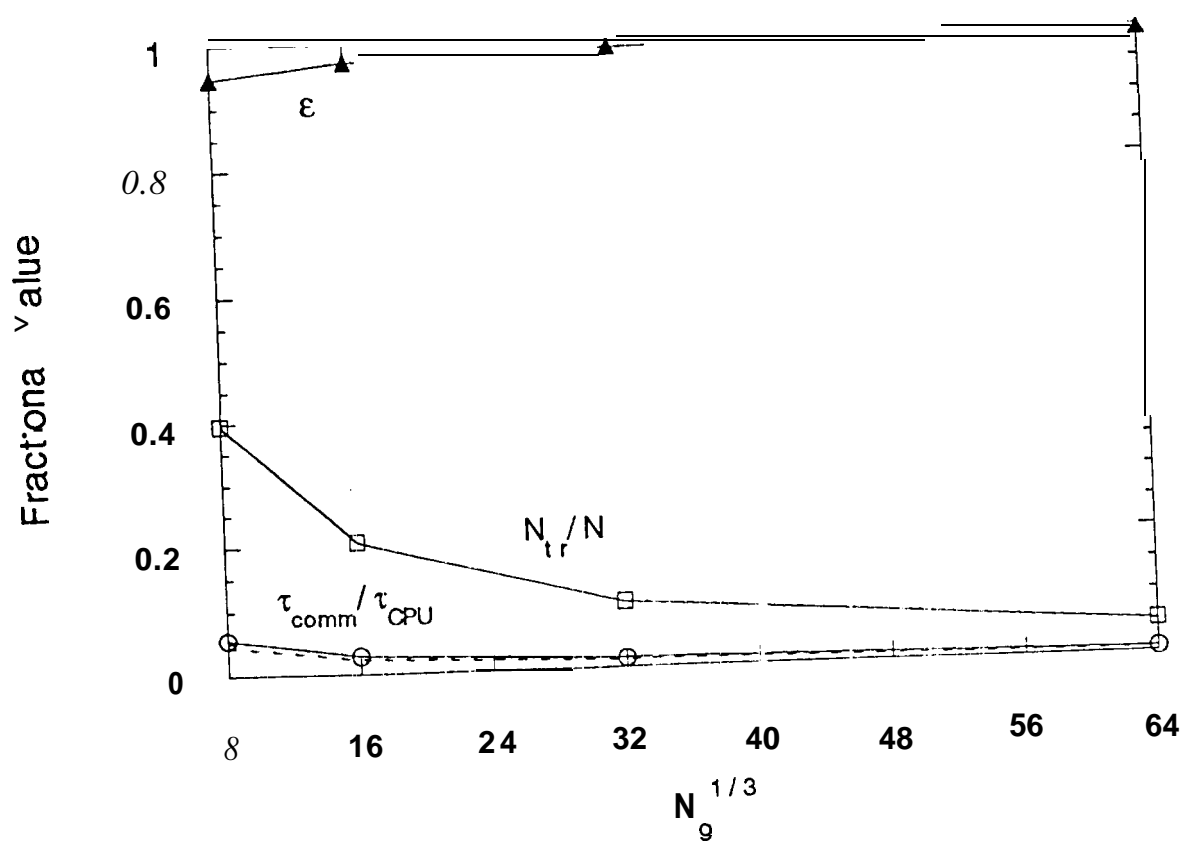


Figure 6

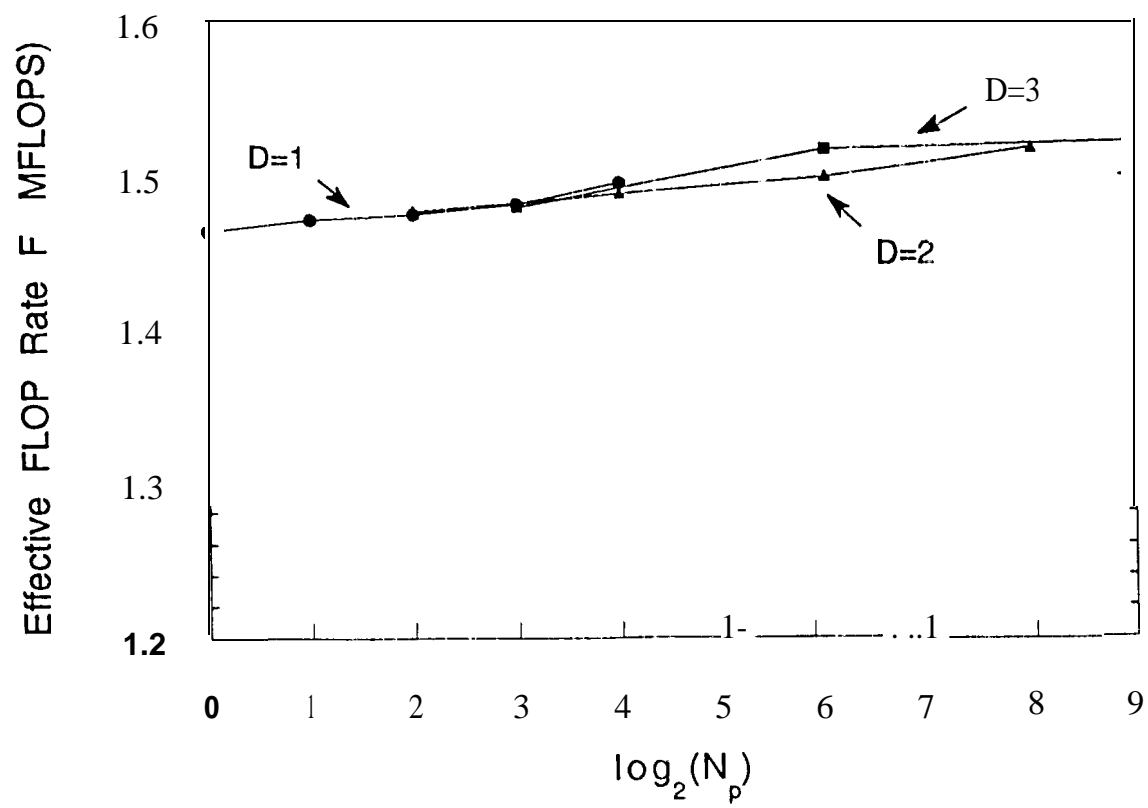


Figure 7

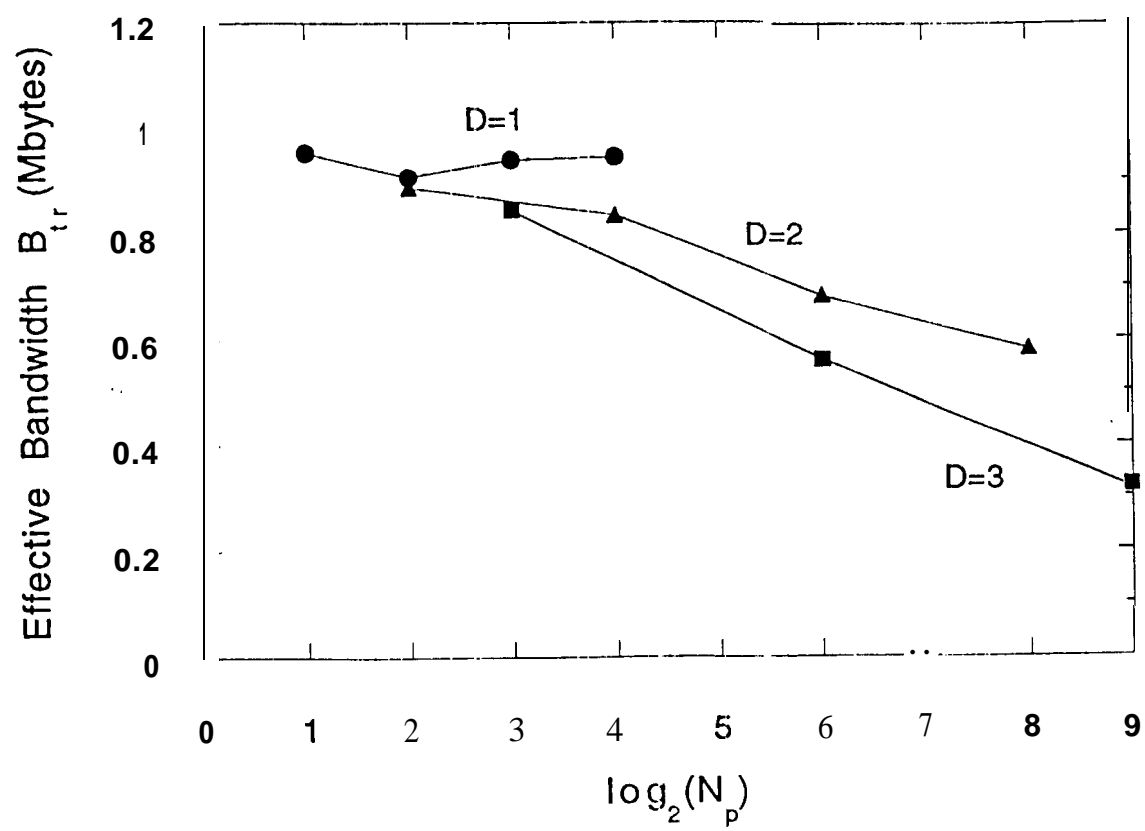


Figure 8 (a)

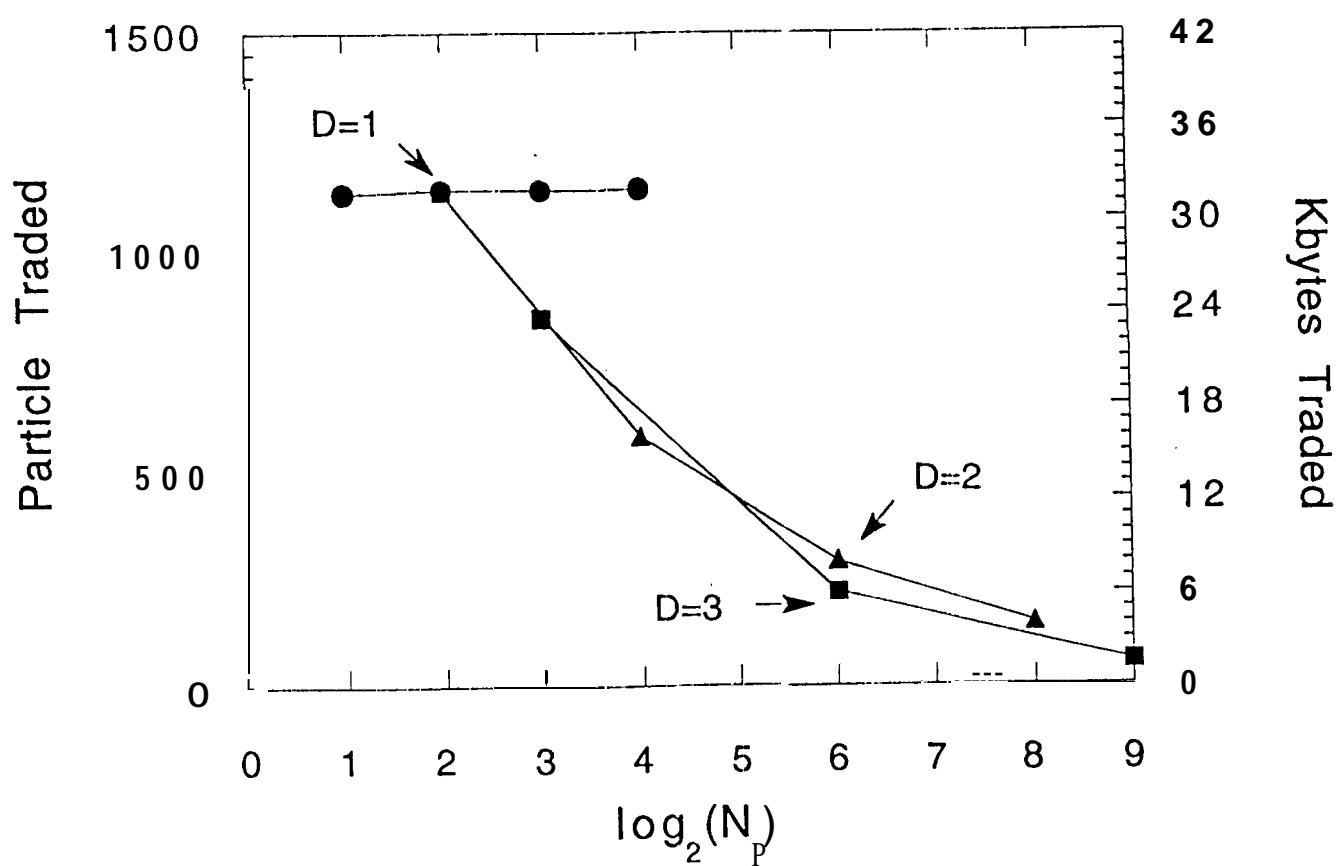


Figure 8 (b)

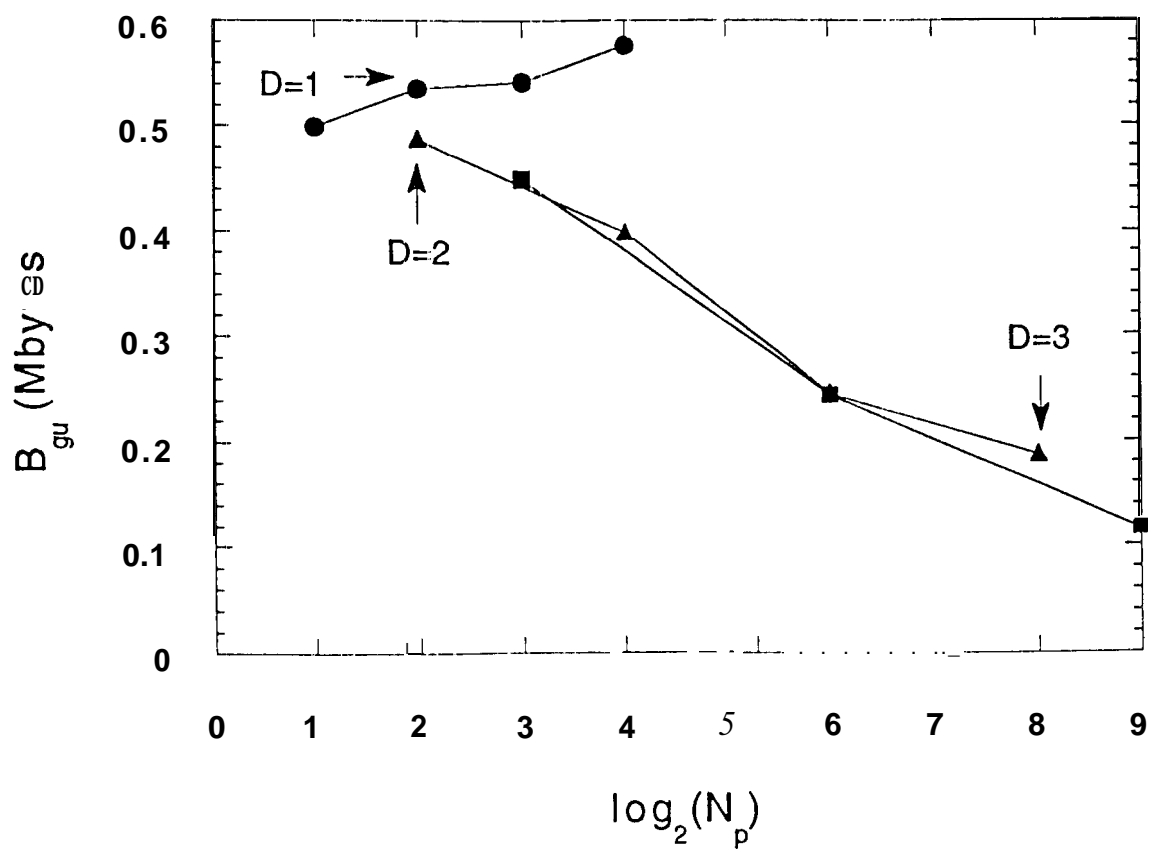


Figure 9 (a)

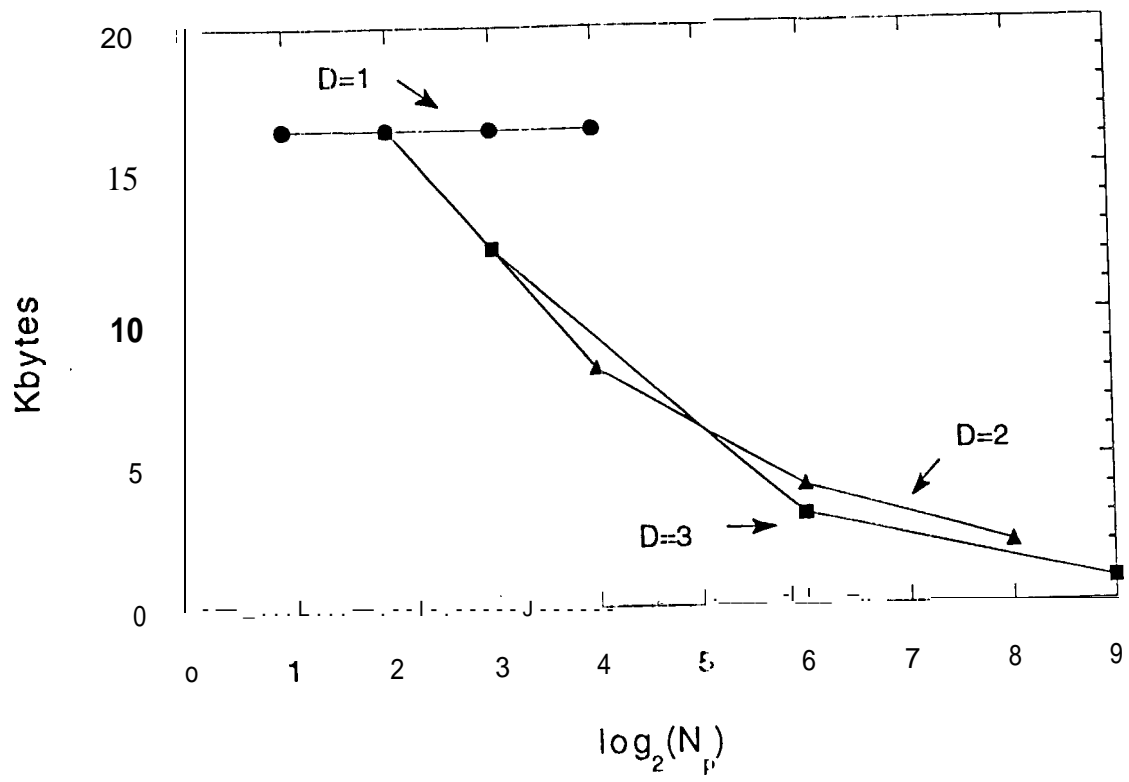


Figure 9 (b)

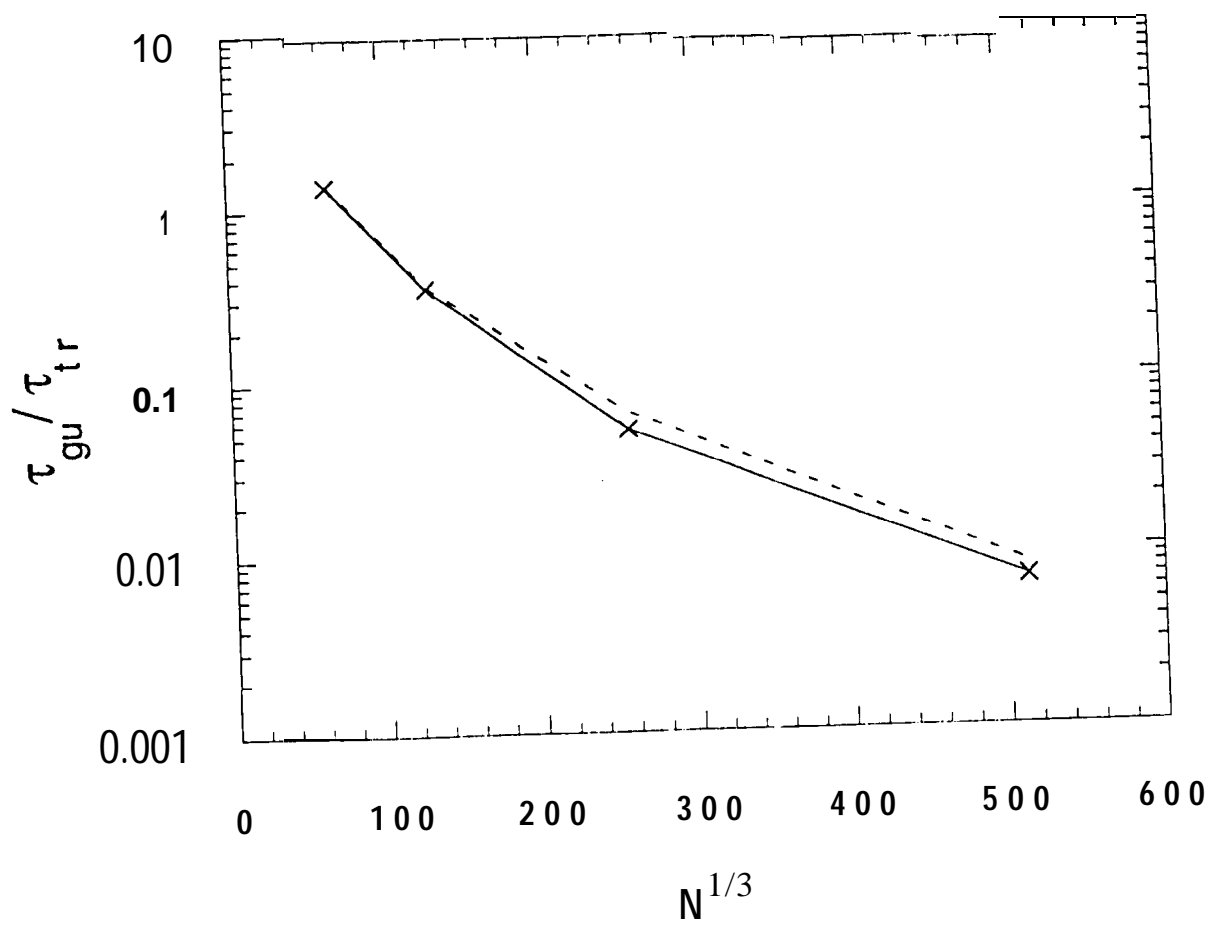


Figure 10